Energy Management in Mobile Devices with the Cinder Operating System

Citation: Arjun Roy, Stephen M. Rumble, Ryan Stutsman, Philip Levis, David Mazières, and Nickolai Zeldovich. Energy management in mobile devices with the cinder operating system. In Proceedings of the sixth conference on Computer systems (EuroSys '11). ACM, New York, NY, USA, 139-152.

As Published: http://dx.doi.org/10.1145/1966445.1966459

Publisher: Association for Computing Machinery (ACM)

Persistent URL: http://hdl.handle.net/1721.1/73663

Version: Author's final manuscript: final author's manuscript post peer review, without publisher's formatting or copy editing

Energy Management in Mobile Devices with the Cinder Operating System

Arjun Roy, Stephen M. Rumble, Ryan Stutsman, Philip Levis, David Mazières, Nickolai Zeldovich†
Stanford University and MIT CSAIL†

Abstract

We argue that controlling energy allocation is an increasingly useful and important feature for operating systems, especially on mobile devices. We present two new low-level abstractions in the Cinder operating system, *reserves* and *taps*, which store and distribute energy for application use. We identify three key properties of control – *isolation*, *delegation*, and *subdivision* – and show how using these abstractions can achieve them. We also show how the architecture of the HiStar information-flow control kernel lends itself well to energy control. We prototype and evaluate Cinder on a popular smartphone, the Android G1.

Categories and Subject Descriptors D.4.7 [*Operating Systems*]: Organization and Design

General Terms Design

Keywords energy, mobile phones, power management

1. Introduction

In the past decade, mobile phones have emerged as a dominant computing platform for end users. These very personal computers depend heavily on graphical user interfaces, always-on connectivity, and long battery life, yet in essence run operating systems originally designed for workstations (Mac OS X/Mach) or time-sharing systems (Linux/Unix).

Historically, operating systems have had poor energy management and accounting. This is not surprising, as their APIs standardized before energy was an issue. For example, the first commodity laptop with performance similar to a desktop, the Compaq SLT/286 [Com 1988], was released just one year before the C API POSIX standard. The resulting energy management limitations of POSIX have prompted a large body of research, ranging from CPU scheduling [Flautner 2002] to accounting [Zeng 2003] to offloading networking. Despite this work, current systems still provide little, if any, application control or feedback: users have some simple high-level sliders or toggles.

This limited control and visibility of energy is especially problematic for mobile phones, where energy and power define system lifetime. In the past decade, phones have evolved from low-function proprietary applications to robust multiprogrammed systems with applications from thousands of sources. Apple announced that as of April 2010 their App Store houses 185,000 apps [App 2010] for the iPhone with more than 4 billion application downloads. This shift away from single-vendor software to complex application platforms means that the phone's software must provide effective mechanisms to manage and control energy as a resource. Such control will be even more important as the danger grows from buggy or poorly designed applications to potentially malicious ones.

In the past year, mobile phone operating systems began providing better support for understanding system energy use. Android, for example, added a UI that estimates application energy consumption with system call and event instrumentation, such as processor scheduling and packet counts. This is a step forward, helping users understand the mysteries of mobile device lifetime. However, while Android provides improved *visibility* into system power use, it does not provide *control*. Outside of manually configuring applications and periodically checking battery use, today's systems cannot do something as simple as controlling email polling to ensure a full day of device use.

This paper presents Cinder, a new operating system designed for mobile phones and other energy-constrained computing devices. Cinder extends the HiStar secure kernel [Zeldovich 2006] to provide new abstractions for controlling and accounting for energy: reserves and taps. *Reserves* are a mechanism for resource delegation, providing fine-grained accounting and acting as an allotment from which applications draw resources. Where reserves describe a quantity of a resource, *taps* place rate limits on resources flowing between reserves. By connecting reserves to one another, taps allow resources to flow to applications. Taps and reserves compose

EuroSys'11, April 10–13, 2011, Salzburg, Austria.

together to allow applications to express their intentions, enabling policy enforcement by the operating system.

Cinder estimates energy consumption using standard device-level accounting and modeling [Zeng 2002]. HiStar's explicit information flow control allows Cinder to track which parties are responsible for resource use, even across interprocess communication calls serviced in other address spaces. Without needing any additional state or support code, Cinder can accurately amortize costs across principals, such as the energy cost of turning on the radio to multiple applications that simultaneously need Internet access.

While Cinder runs on a variety of hardware platforms (AMD64, i386, ARM), the most notable is the HTC Dream, a.k.a. the Android G1. To the best of our knowledge, other than extensions to Linux, Cinder is the first research operating system that runs on a mobile phone. The reason for such a first is simple: the closed nature of phone platforms makes porting an operating system exceedingly difficult.

This paper makes three research contributions. First, it proposes reserves and taps as new operating system mechanisms for managing and controlling energy consumption. Second, it evaluates the effectiveness and power of these mechanisms in a variety of realistic and complex application scenarios running on a real mobile phone. Third, it describes experiences in writing a mobile phone operating system, outlining the challenges and impediments faced when conducting systems research on the dominant end-user computing platform of this decade.

2. A Case for Energy Control

This section motivates the need for low-level, fine-grained energy control in a mobile device operating system. It starts by reviewing some of the prior work on energy visibility and the few examples of coarse energy control. Using several application examples as motivation, it describes three mechanisms an OS needs to provide for energy: isolation, delegation, and subdivision. The next section describes reserves and taps, abstractions which provide these mechanisms at a fine granularity.

2.1 Prior Work on Visibility and Control

Managing energy requires accurately measuring its consumption. A great deal of prior work has examined this problem for mobile systems, including ECOSystem [Zeng 2002], Currentcy [Zeng 2003], PowerScope [Flinn 1999b], and PowerBooter [Zhang 2010]. These systems use a model of the power draw of hardware components based on hardware states. For example, an 802.11b card draws only slightly more power while transmitting than receiving, whereas a CPU's power draw increases with utilization. Current mobile phone energy accounting systems, such as Android's, use this approach. Cinder also does as well; Section 4 provides the details.

Early systems like ECOSystem [Zeng 2002] proposed mechanisms by which a user could control per-application energy expenditure. ECOSystem, in particular, introduced an abstraction called Currentcy, which gives an application the ability to spend a certain amount of energy, up to a fixed cap. This flat hierarchy of energy principals – applications – is reasonable for simple large applications. Mobile applications and systems today, however, are far more complex and involve multiple principals. For example, web browsers run active code as well as possibly untrusted plugins, network daemons control access to the cellular data network, and peripherals have complex energy profiles.

2.2 Isolation, Delegation, and Subdivision

We believe that for applications to effectively control energy, an operating system must provide three energy management mechanisms: *isolation*, *delegation*, and *subdivision*. We motivate these mechanisms through application examples that we follow through the rest of the paper.

The first mechanism is isolation. Isolation is a fundamental part of an operating system. Memory and inter-process communication (IPC) isolation provide security, while CPU and disk space isolation ensure that processes cannot starve others. Isolating energy consumption is similarly important. An application should not be permitted to consume inordinate amounts of energy, nor should it be able to deprive other applications. Consider two processes in a system, each with some share of system energy. To improve system reliability and simplify system design, the operating system should isolate each process' share from the other's. If one process forks additional processes, these children must not be able to consume the energy of the other.

The second mechanism is delegation. Delegation allows a principal to loan any of its available energy and power to another principal. After delegation, either the resource donor or the recipient can freely consume the delegated resources. Furthermore, if there are multiple donors delegating to this recipient, the resources are pooled for use by the recipient. Resource delegation is an important enabler of inter-application cooperation. For example, the Cinder *netd* networking stack transfers energy into a common radio activation pool when an application cannot afford the high initial expense of powering up the radio. By delegating their energy to the radio, multiple processes can contribute to expensive operations; this may not only improve quality of service, but even reduce energy consumption.

The third mechanism is subdivision. Subdivision allows applications to partition their available energy. Combined with isolation, subdivision allows an application to give another principal a partial share of its energy, while being assured that sure that the rest will remain for its own use. For example, modern web browsers commonly run plugins, some of which may even be untrusted. If a browser is granted a finite amount of power, it might want to protect itself from buggy or poorly written plugins that could waste CPU en-

ergy. Subdivision lets the browser give full control over a fraction of its energy allotment to plugins. Isolation further ensures that each plugin component does not consume more than its share.

2.3 Prior Systems

Prior systems like ECOSystem [Zeng 2002, 2003] only partially support isolation and subdivision: child processes share the resources of their parent. This is sufficient when applications are static entities, but not when they spawn new processes and invoke complex services. The web browser demonstrates the problem: it has no way to prevent its plugins from consuming its own resources once they are spawned. Cinder's subdivision lends naturally to familiar and standard abstractions such as process trees, resource containers, and quotas.

Furthermore, prior systems do not permit delegation, which is akin to priority inheritance. For always-on systems which have small variations in power draw, such as the laptops for which they were designed, this is not a serious limitation. On mobile phones, however, which have almost two orders of magnitude difference in active and sleep power, the cost of powering up peripherals, such as the wireless data interface, can be significant. Delegation provides a means to facilitate application cooperation.

3. Design

Cinder is based on HiStar [Zeldovich 2006], a secure operating system built upon information flow control. Cinder adds two new fundamental kernel object types: *reserves* and *taps*. This section gives a brief overview of HiStar and key features related to resource management, describes reserves and taps, gives examples of how they can be used, and details how they are secured.

3.1 HiStar

HiStar is composed of six first-class kernel objects, all protected by a security *label*. Its segments, threads, address spaces, and devices are similar to those of conventional kernels. *Containers* enable hierarchical control over deallocation of kernel objects – objects must be referenced by a container or face garbage collection. *Gates* provide protected control transfer of a thread from one address space to a named offset in another; they are the basis for all IPC.

3.2 Reserves

A reserve describes a right to use a given quantity of a resource, such as energy. When an application consumes a resource the Cinder kernel reduces the values in the corresponding reserve. The kernel prevents threads from performing actions for which their reserves do not have sufficient resources. Reserves, like all other kernel objects, are protected by a security label (§3.5) that controls which threads can observe, use, and manipulate it.

All threads draw from one or more energy reserves. Cinder's CPU scheduler is energy-aware and allows a thread to run only when at least one of its energy reserves is not empty. Threads that have depleted their energy reserves cannot run. Tying energy reserves to the scheduler prevents new spending, which is sufficient to throttle energy consumption.

Reserves allow threads to delegate and subdivide resources. As a simple example, an application granted 1000 mJ of energy can subdivide its reserve into an 800 mJ and a 200 mJ reserve, allowing another thread to connect to the 200 mJ reserve. However, threads rarely manage energy in such concrete quantities, preferring instead to use taps (§3.3). A thread can also perform a reserve-to-reserve transfer provided it is permitted to modify both reserves.

Reserves also provide accounting by tracking application resource consumption. Applications may access this accounting information in order to provide energy-aware features. Finally, reserves can be deleted directly or indirectly when some ancestor of their container is deleted, just as a file can be deleted either directly or indirectly when a directory containing it is deleted in a Unix system.

3.3 Taps

A tap transfers a fixed quantity of resources between two reserves per unit time, which controls the maximum rate at which a resource can be consumed. For example, an application reserve may be connected to the system battery via a tap supplying 1 mJ/s (1 mW).

Taps aid in subdividing resources between applications since partitioning fixed quantities is impractical for most policies. A user may want her phone to last at least 5 hours if she is surfing the web; the amount of energy the browser should receive is relative to the length of time it is used. Providing resources as a rate naturally addresses this.

Another approach, which Cinder does not take, would be to implement transfer rates between reserves through threads that explicitly move resources and enforce rate-limiting as well as accounting. Given five applications, each to be limited to consume an average of 1 W, the system could create five application reserves and threads, with each thread transferring while tracking and limiting energy into each of these applications' reserves. However, this fine-grained control would cause a proliferation of these special-purpose threads, adding overhead and decreasing energy efficiency.

Taps are made up of four pieces of state: a rate, a source reserve, a sink reserve, and a security label containing the privileges necessary to transfer the resources between the source and sink (§3.5). Conceptually, it is an efficient, special-purpose thread whose only job is to transfer energy between reserves. In practice, transfers are executed in batch periodically to minimize scheduling and context-switch overheads.

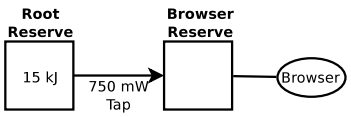

Figure 1. A 15 kJ battery, or *root reserve*, connected to a reserve via a tap. The battery is protected from being misused by the web browser. The web browser draws energy from an isolated reserve which is fed by a 750 mW tap.

3.4 Resource Consumption Graph

Reserves and taps form a directed graph of resource consumption rights. The root of the graph is a reserve representing the system battery; all other reserves are a subdivision of this root reserve. Figure 1 shows a simple example of a web browser whose consumption is rate limited using a tap. The tap guarantees that even if the browser is aggressively using energy the battery will last at least 5 hours (15,000 J at 0.750 J/s is about 5.6 hours).

3.5 Access Control & Security

Any thread can create and share reserves or taps to subdivide and delegate its resources. This ability introduces a problem of fine-grained access control. To solve this, reserves and taps are protected by a security label, like all other kernel objects. The label describes the privileges needed to observe, modify, and use the reserve or tap.

Using resources from a reserve requires both observe and modify privileges: observe because failed consumption indicates the reserve level (zero) and modify for when consumption succeeds. Since a tap actively moves resources between a source and sink reserve, it needs privileges to observe and modify both reserve levels; to aid with this, taps can have privileges embedded in them.

4. Cinder on the HTC Dream

Controlling energy requires measuring or estimating its consumption. This section describes Cinder's implementation and its energy model. The Cinder kernel runs on AMD64, i386, and ARM architectures. All source code is freely available under open-source licenses. Our principal experimental platform is the HTC Dream (Google G1), a modern smartphone based on the Qualcomm MSM7201A chipset.

4.1 Energy accounting

Energy accounting on the HTC Dream is difficult due to the closed nature of its hardware. It has a two-processor design, as shown in Figure 2. The operating system and applications run on an ARM11 processor. A secure, closed ARM9 coprocessor manages the most energy hungry, dynamic, and informative components (e.g. GPS, radio, and battery sensors). The ARM9, for example, exposes the battery level as an integer from 0 to 100.

Recent work on processors has shown that fine-grained performance counters can enable accurate energy estimates

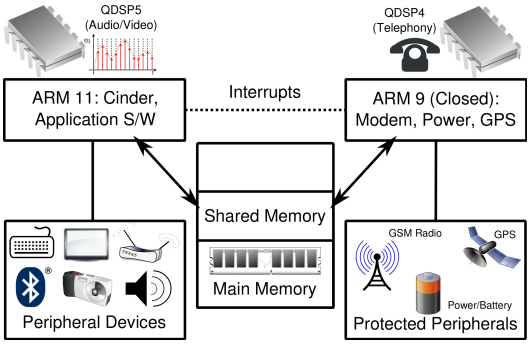

Figure 2. The two ARM cores in the MSM7201A chipset. Cinder runs on the ARM11, whereas the ARM9 controls access to sensitive hardware including the radio and GPS. The two communicate via shared memory and interrupt lines.

within a few percent [Economou 2006; Snowdon 2009]. Without access to such state in the HTC Dream, however, Cinder relies on the simpler well-tested technique of building a model from offline-measurements of device power states in a controlled setting [Flinn 1999b; Fonseca 2008; Zeng 2002]. Phones today use this approach, and so Cinder has equivalent accuracy to commodity systems.

4.2 Power Model

Our energy model uses device states and their duration to estimate energy consumption. We measured the Dream's energy consumption during various states and operations. All measurements were taken using an Agilent Technologies E3644A, a DC power supply with a current sense resistor that can be sampled remotely via an RS-232 interface. We sampled both voltage and current approximately every 200 ms, and aggregated our results from this data.

While idling in Cinder, the Dream uses about 699 mW and another 555 mW when the backlight is on. Spinning the CPU increases consumption by 137 mW. Memory-intensive instruction streams increase CPU power draw by 13% over a simple arithmetic loop. However, the HTC Dream does not have hardware support to estimate what percentage of instructions are memory accesses. The ARM processor also lacks a floating point unit, leaving us with only integer, control flow, and memory instructions. For these reasons, our CPU model currently does not take instruction mix into account and assumes the worst case power draw (all memory intensive operations).

4.3 Peripheral Power

The baseline cost of activating the radio is exceptionally high: small isolated transfers are about 1000 times more expensive, per byte, than large transfers. Figure 3 demonstrates the cost of activating the radio and sending UDP packets to an echo server that returns the same contents. Results demonstrate that the overhead involved dominates the total

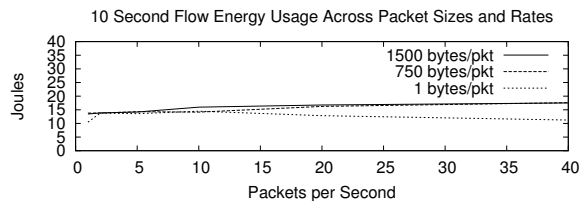

Figure 3. Radio data path power consumption for 10 second flows across six different packet rates and three packet sizes. Short flows are dominated by the 9.5 J baseline cost shown in Figure 4. For this simple static test, data rate has only a small effect on the total energy consumption. The average cost is 14.3 J (minimum: 10.5, maximum: 17.6).

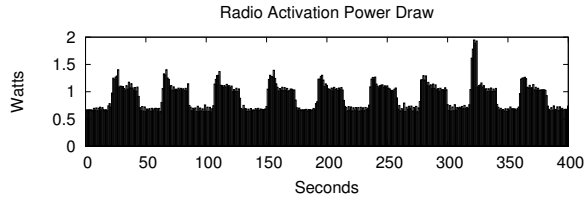

Figure 4. Cost of transitioning from the lowest radio power state to active. One UDP packet is transmitted approximately every 40 seconds to enable the radio. The device fully sleeps after 20 seconds, but the average plateau consumes an additional 9.5 J of energy over baseline (minimum 8.8 J, maximum 11.9 J). Power consumption for a stationary device can often be predicted with reasonable accuracy, but outliers, such as the penultimate transition, occur unpredictably.

power cost for flows lasting less than 10 seconds in duration, regardless of the bitrate.

Figure 4 shows this activation cost. An application powers up the radio by sending a single 1-byte UDP packet. The secure ARM9 automatically returns to a low power mode after 20 seconds of inactivity. Because the ARM9 is closed, Cinder cannot change this inactivity timeout.

With this workload, it costs 9.5 joules to send a single byte! One lesson from this is that coordinating applications to amortize energy start-up costs could greatly improve energy efficiency. In §5.5 we demonstrate how Cinder can use reserves and taps for exactly this purpose.

4.4 Mobility & Power Model Improvements

Cinder's aim is to leverage advances in energy accounting (see §8.2) to allow users and applications to provision and manage their limited budgets. Accurate energy accounting is an orthogonal and active area of research. Cinder is adaptable and can take advantage of new accounting techniques or information exposed by device manufacturers.

```
// Create a reserve
object_id_t res_id;
res_id = reserve_create(container_id, res_label);
objref res = OBJREF(container_id, res_id);

// Create a tap and connect it between
// the battery and the new reserve
object_id_t tap_id;
tap_id = tap_create(container_id, root_reserve,
                    res, tap_label);
objref tap = OBJREF(container_id, tap_id);
// Limit the child to 1 mW
tap_set_rate(tap, TAP_TYPE_CONST, 1);

if (fork() == 0) {
    // child process: switch to new reserve before exec
    self_set_active_reserve(res);
    execv(args[0], args);
}
```

Figure 5. `energywrap` excerpt without error handling.

5. Applications

To gain experience with Cinder's abstractions, we developed applications using reserves and taps. This section describes these applications, including a command-line utility that augments existing applications with energy policies, an energy constrained web browser that further isolates itself from its browser plugins, and a task manager application that limits energy consumption of background applications.

5.1 `energywrap`

Taking advantage of the composability of Cinder's resource graph, the `energywrap` utility allows any application to be sandboxed even if it is buggy or malicious. `energywrap` takes a rate limit and a path to an application binary. The utility creates a new reserve and attaches it to the reserve in which `energywrap` started by a tap with the rate given as input. After forking, `energywrap` begins drawing resources from the newly allocated reserve rather than the original reserve of the parent process and executes the specified program. This allows even energy-unaware applications to be augmented with energy policies.

The sandboxing policy provided by `energywrap` is implemented in about 100 lines of C++. An excerpt is shown in Figure 5. HiStar provides a `wrap` utility designed to isolate applications with respect to privileges and storage resources. Coupling this utility with `energywrap` allows any application or user to provide a virtualized environment to any thread or application. Section 6.1 evaluates the effectiveness of energy sandboxing and isolation.

`energywrap` has proved useful in implementing policies while designing and testing Cinder, particularly for legacy applications that have no notion of reserves or taps. Since `energywrap` runs an arbitrary executable, it is possible to use `energywrap` to wrap itself or shell scripts, which may invoke `energywrap` with other scripts or applications. This

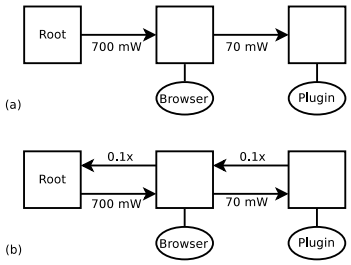

Figure 6. (a) A web browser configured to run for at least 6 hours on a 15 kJ battery. The web browser further ensures that its plugin cannot use more than 10% of its energy. (b) Adding 0.1x backward proportional taps promotes sharing of excess energy unused by the browser and plugin.

allows a wide class of ad hoc policies to be scripted using standard shell scripting or on-the-fly at the command line.

5.2 Fine-grained Control

Mobile browsers now support plugins like Adobe Flash [Fla 2009], and we can expect more plugins and extensions to follow. On a device where resources are precious, it is important to have tight control over these plugins.

In Cinder, an application may be given some fixed rate or quota of energy using reserves and taps. A web browser may, for example, want to also run a plugin while ensuring that it cannot starve other plugins or even the browser itself. Shown in Figure 6a, the browser can allocate a separate reserve for the plugin and connect it to its own energy via a low rate tap.

Often a single plugin (e.g. Flash) may be handling a number of pages or requests all in a single process. To scale the energy given to the plugin with the number of pages it is handling, the browser can add a tap per page. When a particular page is no longer being handled (e.g. the user navigates away) the taps associated with that page can be automatically garbage collected, effectively revoking those power sources.

Cinder includes a simple graphical web browser based on links2 that runs in Xorg or standalone against the framebuffer. It is augmented with an extension running in a separate process, whose energy usage is subdivided and isolated from the browser. The browser can send requests to the extension process (for ad blocking, etc.), and if the extension is unresponsive due to lack of energy the browser can display the unaugmented page.

5.2.1 Reclaiming Unused Resources

Consider a problem common to many applications: a web browser would like to allow a plugin to consume resources quickly while making sure it shares unused resources. The plugin may fully utilize peripherals and drive the device at peak power, requiring a reserve fed with a high rate tap. This raises a problem: if the plugin draws less than its tap rate, the reserve will slowly fill with energy that no other application can use.

To solve this problem, an application can use a proportional tap. These taps transfer a fraction of their source reserve's resource per unit time, rather than a fixed quantity. Figure 6b shows the fix to the browser; the plugin reserve on the right is limited to a maximum average power draw of 70 mW. The backwards proportional tap means the plugin reserve can store up to 10 s of this power (700 mJ) for bursty operations. Once the reserve reaches 700 mJ, the backwards proportional tap drains the reserve as quickly as the forward constant tap fills it. Similarly, the browser's reserve can accumulate up to 7000 mJ while being forced to share unused energy with other applications.

5.2.2 Hoarding and Resource Decay

Backward proportional taps alone are insufficient for preventing malicious applications from hoarding. Threads can sidestep taxation by creating a new reserve with no proportional taps and periodically transferring resources to it. The application could, over time, accumulate energy equal to the battery and starve the rest of the system.

To prevent this, Cinder could provide a `reserve_clone()` rather than `reserve_create()`. This call would take a reserve that an application has access to and create a new reserve taking care to duplicate any backward proportional taps that the application does not have the permission to remove. Additionally, Cinder would need to disallow system calls that transfer resources from a fast-draining reserve to a more slow-draining reserve unless the caller has proper permission (that is, the permission to remove all the backward taps from the source reserve that do not have a corresponding backward tap at the target reserve).

These constraints eliminate hoarding, but complicate applications that are not malicious. Therefore, in practice, Cinder prevents hoarding by imposing a global, long-term decay of resources across all reserves; every reserve has an implicit proportional backward tap to the battery.

By default, Cinder is configured to leak 50% of reserve resources after a period of 10 minutes. This long (but short compared to the period between battery recharges) half-life allows applications to accumulate and store energy for significant periods, and permits the system to make large-scale long-term hoarding impossible. ESX Server [Waldspurger 2002] successfully uses a similar "idle memory tax" to mitigate hoarding of unused memory between virtual machines.

Further experience with these abstractions is needed to understand whether the trade-offs associated with the more fundamental solution for hoarding are worth making.

5.3 Energy-Aware Applications

Using Cinder, developers can gain fine-grained control of resources within their applications, providing a better experience to end users. This includes adaptive policies for programs where partial or degraded results are still useful, and

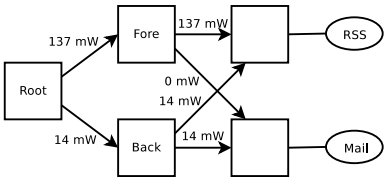

Figure 7. `RSS` is running in the foreground so the task manager has set its tap to give it additional power. `Mail` is running in the background, and can only draw energy from the background reserve. This ensures that actual battery consumption matches the user's expectation that the visible application is responsible for most energy consumption.

offer a compromise between battery life and user experience. For example, smart applications may scale the quality of streaming video or reduce texture quality in a game when available energy is low, since the user can still watch a video or play a game when insufficient resources are available to run at full fidelity.

As a concrete example, we have implemented an energy-aware network picture gallery. The application has a separate thread for downloading images, using an energy reserve distinct from the main thread. The rate the application consumes energy from this reserve depends on the frequency of image requests and the requested image sizes. The application checks the levels in the reserve periodically. A drop in the reserve level indicates that the downloader is consuming energy too quickly and will be throttled if it cannot curb consumption. In this case, the downloader only requests partial data from the remote interlaced PNG images, which yields a lower quality image in exchange for reduced data transfer over the network (and lower consumption by the device). Section 6.2 evaluates the effectiveness of these adaptations.

5.4 Background Applications

Background applications complicate resource management. Despite being invisible to the user, an application may be using resources. This discontinuity between reality and user perception makes the user suspicious of foreground applications they have used frequently, which may not be responsible. Cinder provides not only a means to understand which applications are using resources, but also a means to manage those resources to meet user expectations. Since the user naturally suspects foreground applications of using energy, he can easily manage his use of those applications. Cinder's job, then, is to manage background applications to prevent them from interfering with the user's natural intuition.

Figure 7 shows how Cinder accomplishes this. Each application has a reserve from which it draws energy. Each such application's reserve is then connected to two other reserves via taps. The first is the foreground reserve, which is connected to the battery via a high rate tap. The second is a low rate reserve connected to the battery via a low rate tap. An application's tap to the background reserve always allows energy to flow; however, the foreground tap is set to a rate of 0 while the application is running in the background, and is set to a high value when the application is running in the foreground. The task manager is the creator of the tap connecting the application to the foreground reserve and, by default, is the only thread privileged to modify the parameters on the tap. Since programs are confined to low power while in the background, the user's expectations are respected. Section 6.3 evaluates this configuration.

5.5 Cooperative Network Stack

Some of the most energy-hungry devices on a mobile platform have complex, non-linear power models (e.g. the data path and the GPS). Careful control over how applications use such devices can result in energy savings. Section 4 shows that the radio has a high initial cost and a much smaller amortized price for bulk transfers. This power profile is a problem for some periodic background applications like email checkers, RSS feed downloaders, weather widgets, and time synchronization daemons. Cinder's network stack, *netd*, improves energy efficiency for this typical class of applications through using two mechanisms: precise resource accounting across process boundaries and flexible sharing and resource transfer control.

5.5.1 Accurate Accounting via Gates

To accurately track which threads cause resource consumption, Cinder uses HiStar's gates, which form the basis of inter-process communication. A gate is a named entry point in an address space, typically corresponding to a daemon or system service available over IPC. Unlike traditional IPC, in which a thread in a client process sends a message to a thread in a server process, here the calling thread itself enters the server's address space.

Since Cinder tracks resource consumption by the active reserve of a thread, the caller of a system-wide service, like *netd*, is billed for resource consumption it causes, even while executing in the other address space. Other systems, such as Linux, would need some form of message tracking during inter-process communication in order to heuristically bill the principals for resource consumption, whereas Cinder provides accurate accounting naturally. Section 7.1 details the complications that arose in reproducing Cinder atop Linux.

5.5.2 Encouraging Cooperation

To facilitate sharing, *netd* contains a reserve where threads cooperatively save up energy for a radio power up event. For each thread that makes a network system call, if the sum of its own reserve and *netd*'s reserve are not sufficient for the power on, the call blocks, contributes the energy acquired by its taps to the *netd* reserve, and sleeps to accumulate more. When there is sufficient energy to turn the radio on and perform the transmissions requested by the waiting threads, Cinder debits the reserve and permits the threads to proceed. The *netd* reserve is not subject to the system global half-

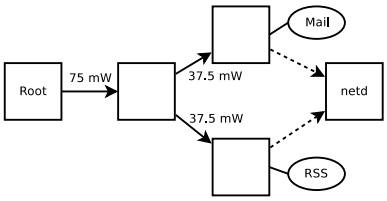

Figure 8. The mail checker and RSS feed downloader are constrained to use up up to 37.5 mW apiece. When making network requests, `netd` explicitly transfers energy from their reserves into its own reserve. Once the requesting application's reserve, combined with the `netd` reserve, has enough energy, the radio will turn on. This simple policy helps synchronize applications' network access, reducing active radio time and saving energy.

life, as the process is trusted not to hoard energy and, by construction, only stores enough energy to activate the radio before being expended.

Cinder estimates the cost of radio access by tracking when network transmit and receive events occur. For instance, if the radio has been idle for 20 seconds or more, threads wishing to use the network must contribute enough energy to turn the radio on and maintain the active power state until it idles again (§4). Once the radio is on, back-to-back actions are cheaper than ones with more delay between them because they extend the active period (delay the next idle period) less significantly.

For example, if the radio has been active for one second, it will automatically idle again 19 seconds later, so transmitting now only extends the active period by 1 second. However, if the radio is active but no packets have been sent or received for 15 seconds, transmitting now will extend the active period by an additional 15 seconds – the same action becomes much more expensive.

This leaves the problem of how to charge for incoming packets since energy has already been spent to receive them. To facilitate this, threads can debit their own reserves up to or into debt even if the cost can only be determined after-the-fact. This allows user space accounting; for example, in this case the receiving thread under the control of *netd*'s *send* gate debits its own reserve when packets are delivered to it.

Section 6.4 evaluates the effectiveness of *netd* in aiding cooperation between applications to increase the responsiveness of services while retaining their energy budget.

6. Evaluation

Using the applications described in §5, we evaluate whether Cinder meets the requirements described in Sections 1 and 2: can it control energy, provide visibility into the energy of a running system, and provide subdivision, delegation, as well as isolation? Furthermore, we evaluate whether Cinder can facilitate dynamic energy-aware applications and improve a system's energy efficiency by managing complex devices with non-linear power consumption.

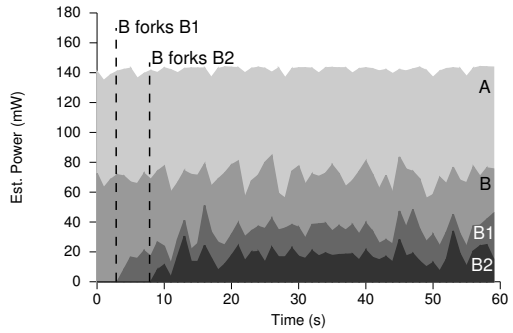

Figure 9. Stacked graph of Cinder's CPU energy accounting estimates during isolated process execution. Process A's energy consumption is isolated from other processes' energy use despite B's periodic spawning of child processes (B1 and B2). The sum of the estimated power of the individual processes closely matches the measured true power consumption of the CPU of about 139 mW during this experiment.

All experiments exception the image viewer of §6.2 use Cinder running on an HTC Dream. The image viewer evaluation was performed on a Lenovo T60p laptop. To measure power draw, we connect the Dream to the Agilent E3644A DC power supply. To monitor reserve energy levels we use the Dream's serial port output.

6.1 Isolation, Subdivision, and Delegation: Buggy and Malicious Applications

We first show how a simple use case – protecting the system from a buggy or malicious energy hog – requires isolation, subdivision, and delegation. Figure 9 shows a stacked plot of Cinder's energy accounting estimates of two processes, A and B. In this experiment, the system is configured to evenly subdivide and delegate enough power to fully utilize the CPU between the two processes (about 68 mW to each process since running the CPU costs 137 mW).

Process B spawns a new child process at about 5 seconds (B1) and again at about 10 seconds (B2). Without reserves and taps, these additional processes would cause A to receive a smaller share of the CPU. Here, however, Process A is isolated from these forks and still consumes about 50% of the CPU (and power share).

This experiment highlights the fine-grained nature of Cinder's control: not only is A isolated from B, but B is also able to protect itself from its own children, B1 and B2. Rather than have its children draw from B's own reserve, B creates two new reserves subdividing and delegating its power to each using two taps. Each of the taps has one-quarter the power of B's tap, such that after spawning both they are using half of B's power. Figure 9 shows that both A and B's policies are composed and enforced in the expected way.

6.2 Subdivision and Delegation: Image Viewer

To demonstrate the practicality of energy-aware applications in Cinder, we used our image viewer described in §5.3,

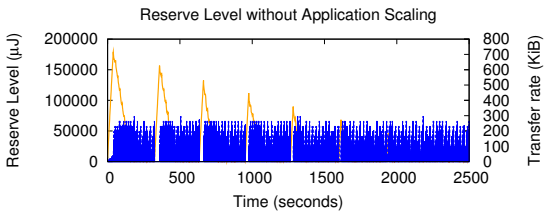

Figure 10. The same image viewer application as in §5.3, but without dynamic scaling of image quality. The line represents energy in the downloader's reserve while the bars represent the amount of data downloaded per image.

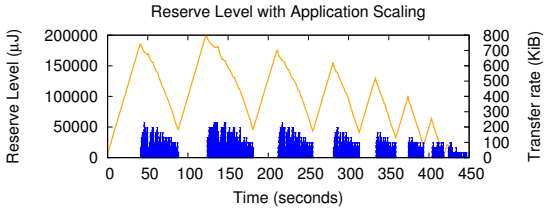

Figure 11. Image viewer with energy-aware scaling of image quality enabled. As energy becomes scarce, quality is lowered and less data is downloaded per image. The experiment takes less than one-fifth the time to complete within the energy budget versus the non-adaptive viewer due to adaptation to reduced available energy.

tested with and without energy-aware image scaling. The tests mimic a user loading a page of images, pausing to view the images, and then requesting more. We tracked the energy reserve levels, the amount of data transferred over the network interface, the download time for each batch of images, and the average bytes transferred per image over time. Each image was of similar size (∼2.7 MiB) and each batch contained the same number of images. Pausing between batches allowed the energy reserve for the downloader thread to fill at a constant rate. The first pause lasted for 40 seconds, with each successive pause being 5 seconds shorter than the previous pause, so a smaller amount of energy built up in the reserve after each batch was downloaded.

When image download sizes are not scaled back as in Figure 10, the amount of data transferred stays constant per batch. With each successive batch, the amount of energy in the reserve at the start of the batch decreases since the user pauses more briefly after downloading. Thus the reserve runs out soon after the start of each batch in this case, with the image transfers stalling until enough energy is available for the thread to continue, causing a long run time.

When image requests use energy-aware scaling as in Figure 11, the quality of images and bytes transferred for each image drops as the energy level dips, and the transfer time per image decreases with the smaller image data. Over the course of the test, the level of energy present in the reserve dropped below the threshold, but never to zero. The images downloaded 5 times more quickly than the viewer which does not scale the images.

6.3 Delegation and Subdivision: Background Applications

Section 5.4 presented a configuration where system power is subdivided into a highly powered task manager reserve and a low powered background reserve. These reserves delegate their energy to applications running in the foreground and background, respectively, allowing background applications to continue to make slow forward progress, but keeping foreground applications responsive. This experiment uses a configuration identical to Figure 7.

Figure 12a shows two processes spinning on the CPU, initially in the background. The background tap provides the two of them 14 mW, enough to keep the 137 mW CPU at 10% utilization. At about 10 seconds, the task manager selects Process A as the foreground process, granting it enough energy to fully utilize the CPU (137 mW). Process B continues to run according to its background power share of 14 mW. At the 20 second mark, the task manager retires Process A to the background by setting its foreground tap rate to 0 mW. At 30 seconds, the task manager gives Process B access to the foreground resources and, similarly, returns it to the background at 40 seconds.

Figure 12b highlights the need for Cinder to prevent large-scale hoarding. The configuration is the same except the foreground tap gives 300 mW of power. Because 300 mW is greater than the CPU cost of 137 mW, applications in the foreground can accumulate excess energy. The two processes move in and out of the foreground as before, but this accumulated energy changes their behavior. When B is moved to the foreground, A still has plenty of energy, and so competes with B for the CPU such that each receives a 50% share. After A exhausts its energy, it returns to its original 14 mW. Shortly thereafter, B moves to the background as well. But just as A did, B accumulated energy during its time in the foreground and so is able to use ∼90% of the CPU until it exhausts its reserve.

The system-wide half-life both caps the total energy hoarding possible during foreground operation and returns applications to the natural background power over a 10 minute period. This allows a process to perform an elevated amount of work briefly after returning to background status if it underutilized its resources while in the foreground.

6.4 Delegation: Cooperative Network Stack

We demonstrate the effectiveness of Cinder's modified *netd* (shown in Figure 8), comparing it to an energy-unrestricted network stack. In both experiments, an RSS feed downloader starts with a poll interval of 60 seconds. Fifteen seconds later, a mail fetcher daemon starts, also with a 60 second poll interval. Both applications are provided enough power to start the radio every 60 seconds, if they work in unison.

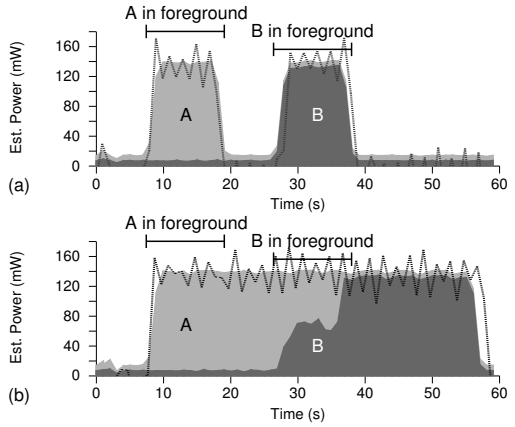

Figure 12. Stacked graph of Cinder's CPU energy accounting estimates as processes A and B spin on the CPU. Together, they are allowed 14 mW while in the background. The task manager runs A in the foreground in the 10 - 20 second interval and B in the foreground during the 30 - 40 second interval. (a) shows the results for the foreground tap providing the process with 137 mW (the precise cost of using the CPU at 100%). (b) shows the foreground tap providing the process with 300 mW. The dotted line shows actual power measurements compensated for baseline power draw with an idle CPU and averaged over 1 second intervals.

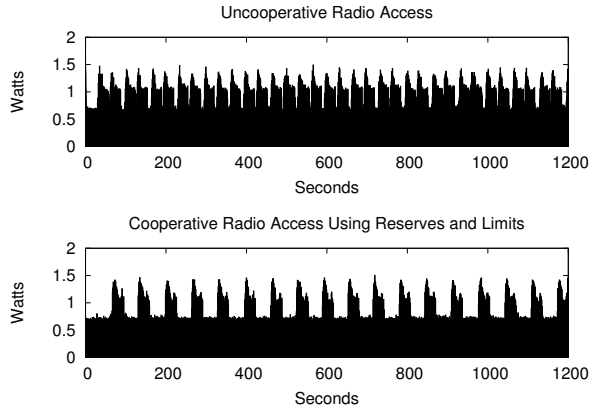

Figure 13. Two background applications, a pop3 mail and an RSS fetcher, each poll every sixty seconds. a) Since they are not coordinated, their use of the radio is staggered, resulting in increased power consumption. Each application uses the radio for at most a few seconds, but neither takes advantage of the other having brought the radio out of the low power idle state. b) The same mail and RSS background applications using reserves and limits to coordinate their access to the radio data path. Enough energy is allocated to each application to turn the radio on every two minutes. By pooling their resources, they are able to turn the radio on at most every sixty seconds.

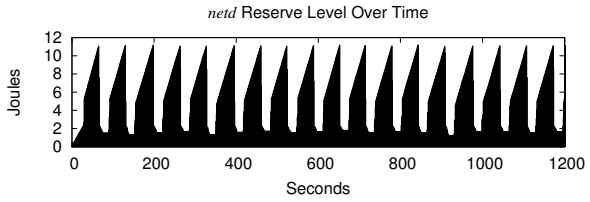

Figure 14. The level of the reserve into which the two background applications transfer their allotted joules. When the reserve reaches a level sufficient to pay for the cost of transitioning the radio to the active state, it is debited, the radio is turned on, and the processes proceed to use the network. Although Figure 4 showed an average 9.5 J cost to power up the radio, *netd* requires 125% of this level before turning the radio on, essentially mandating that applications have extra energy to transmit and receive subsequent packets. Therefore, the reserve does not empty to 0.

	Non-Coop	Coop	Improv
Total Time	1201s	1201s	N/A
Total Energy	1238J	1083J	12.5%
Active Time	949s	510s	46.3%
Active Energy	1064J	594J	44.2%

Table 1. Improvements in energy consumption and active radio time using cooperative resource sharing in Cinder. Energy use due to the radio is reduced, resulting in a 12.5% total system power reduction over the 20 minute experiment.

Figure 13a shows the uncooperative applications wasting energy – running when the radio is idle and powering it up independently. Neither combines efforts to amortize costs.

In comparison, Figure 13b shows what happens with the modified *netd*. Each application still only receives enough energy to activate the radio every two minutes; however, when they initiate network operations, their threads block and contribute acquired energy to the *netd* reserve (Figure 14). Since the two threads combine their power in the *netd* reserve, every 60 seconds enough energy is saved to use the radio and both applications proceed simultaneously.

Using delegation, independent applications in Cinder automatically collaborate, improving quality of service. In this case, the improved quality of service is increasing the frequency of mail and news checks by a factor of two, using the same energy budget. Table 1 shows the energy savings of the modified *netd*. In total, 12.5% less energy is used in the same time interval for an equivalent amount of work. While significant, we stress that our baseline power consumption is artificially dominant, as Cinder does nothing to place the hardware into lower power states while idle in contrast to Linux. We expect Cinder to provide greater improvement on a mature mobile platform that makes full use of power saving features.

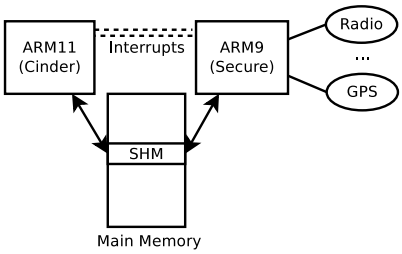

Figure 15. Cinder may only indirectly access many hardware features, such as the radio and GPS, by passing messages to a secure ARM9 coprocessor.

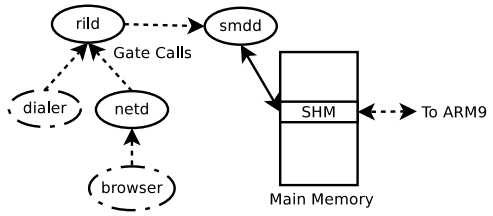

Figure 16. The user-level smdd daemon manages the shared memory interface on the ARM11 and exports interfaces to the radio, GPS, battery sensor, and so on via gate calls. Consumers of this interface, such as the radio daemon, *rild*, may also export their own gate calls. *netd*, for example, implements gates for libOS TCP/IP sockets. Gates are used by both user applications (*browser, dialer*) and OS daemons (*netd*, etc.).

7. Experience Developing on a Mobile Platform

We ported Cinder to the HTC Dream mobile phone. Because developing a kernel for a mobile phone platform is a non-trivial task that is rarely attempted, we describe our process here in detail.

To run Cinder on the HTC Dream, we first ported the kernel to the generic ARM architecture (2,380 additional lines of C and assembly). MSM7201A-specific kernel device support for timers, serial ports, framebuffer, interrupts, GPIO pins, and keypad required another 1,690 lines of C. Cinder implements the GSM/GPRS/EDGE radio functionality in userspace with Android driver ports.

Implementing radio functionality is particularly difficult, as it requires access to secure and undocumented hardware that is not directly accessible from the processor. For instance, the MSM7201A chipset includes two cores: the ARM11 runs application code (Cinder), while a secure ARM9 controls the radio and other sensitive features (Figure 15). Accessing these features requires communicating between the cores using a combination of shared memory and interrupt lines. We first mapped the shared memory segment into a privileged user-level process and ported the Android Linux kernel's shared memory device to userspace (Figure 16). This daemon, *smdd* (4,756 lines), exports ARM9 services via gate calls to other consumers, including the radio interface library (RIL). The RIL generates and consumes messages between cores that initiate and respond to radio events, such as dialing a number or being notified of an incoming call.

In Android, the radio interface library consists of two parts: an open source generic interface library that provides common radio functions across different hardware platforms, and a device-specific, Android-centric shared object that interfaces with specific modem hardware (libril.so). Unfortunately, libril.so is closed-source and precompiled for Android: this makes it excessively difficult to incorporate into another operating system. Without hardware documentation or tremendous reverse engineering, using the radio requires running this shared object in Cinder. To do so, we wrote a compatibility shim layer to emulate both Android's "bionic" libc interface, as well as the various `/dev` devices it normally uses to talk to the ARM9 (1,302 lines of C). We rewrote the library's symbol table to link against our compatibility calls, rather than the binary-incompatible uClibc functions and syscalls that regular Cinder applications use. Finally, we wrote a port of the radio interface library frontend that provides gates to service radio requests from applications needing network access.

Cinder currently supports the radio data path (IP), and can send and receive SMS text messages. Cinder can also initiate and receive voice calls, but as it does not yet have a port of the audio library, calls are silent. In retrospect, since hardware documentation is unavailable, basing our solution on Android, rather than HiStar, would have been far simpler from a device support perspective. Crucially, however, our implementation atop Linux trades the simple and accurate IPC resource accounting needed in energy management for device drivers (§7.1). We felt that a cleaner slate justified the additional tedium as well as the reduced hardware support present in our prototype.

In summary, even trivial radio operation is quite complicated, requiring about 12,000 lines of userspace code along with the 263 KiB closed libril.so. In comparison, the entire Cinder kernel consists of about 27,000 lines of C for all four CPU architectures and all device drivers. The kernel is only 644 KiB – less than 2.5 times the size of libril.so.

7.1 Cinder-Linux versus Cinder-HiStar

Cinder was initially implemented on HiStar because several key behaviors of the platform are naturally expressible using HiStar's abstractions.

One such feature of Cinder is resource delegation between principals. Consider a common situation where a client process P requires work to be performed on its behalf by a daemon process D. A real world example is the radio interface layer daemon on the Android platform. Cinder must ensure P is charged for any work D performs on its behalf – or, equivalently, it must ensure that P provides the resources that D's code uses to run.

HiStar's abstractions achieve this behavior cleanly and simply. A process in HiStar is a container, containing an address space and one or more threads. IPC is performed through special gates defined by the process – a thread belonging to process P can enter a gate defined by process D, after which the thread has access to D's address space, though while under control of D's code text. When process P requires service from daemon D a thread, T, belonging to P enters D's address space via a gate. Cinder debits T for work it performs as usual even though it executes under the control of D's code, correctly billing consumption to P. This way, HiStar's IPC mechanism easily achieves the desired delegation behavior.

Linux, on the other hand, uses several different facilities to provide IPC, many of which are based upon message passing between processes. A few examples are pipes, Unix domain sockets, message queues, and semaphores. These forms of IPC occur without any resource sharing or attribution between processes. This subverts delegation since process P may elicit work by daemon D on its behalf without providing the resources for the work.

To compensate, Linux needs to verify that the calling process has provided adequate resources to perform the desired request. However, existing IPC mechanisms in Linux are not built with the goal of discovering the identity of the caller in mind. Consider a daemon D that reads requests from a named pipe in the filesystem. When D reads from the pipe, it only knows the writing process has permissions to access the pipe. In general, it cannot identify which process in the system made the request, and thus does not know which process to debit.

To mitigate this problem, Cinder-Linux needs a way for the daemon to determine the identity of the calling process. One possibility is to have a user level protocol in which a calling process P encodes both its identity and a description of how D can access resources that P has set aside for D within the request. For example, it could format a request as a triple: $\langle$`pid, reserve_id, request`$\rangle$. D accesses the reserve named in the request, and only performs work once it ensures the caller has provided sufficient resources in payment. Since a user level process can lie abouts it credentials, the protocol is not robust against malicious applications. A more robust mechanism would require new kernel IPC mechanisms.

Both Cinder-HiStar and Cinder-Linux must prevent resource misuse. In particular, D must not co-opt P's resources for performing unrelated tasks, and process P must provide resources for work performed by D on its behalf. Providing these guarantees on Linux requires either a fine grained permissions system or, alternatively, some form of information flow control or tracking (with which the daemon could determine which process sent a given request). In contrast, HiStar's existing information flow control mechanisms easily provide the necessary protection.

Linux has the benefit of being an established operating system with vast device driver support and the entire Android platform. As a result, it is easier to write real-world applications. Consequently, we have written an initial implementation of Cinder that runs on top of the Linux and the Android platform on the Dream. The basics of Cinder-Linux remain the same as Cinder-HiStar aside from resource attribution issues for IPC and fine-grained permissions. Most implementation of the Cinder abstractions are independent of the underlying operating system and similar on HiStar and Linux. Some differences in the implementation do exist, however. For example, Cinder-HiStar flows taps during scheduler timer interrupts, while Cinder-Linux uses a kernel thread. One area of future work is further testing the concepts and features of Cinder on the Cinder-Linux platform.

8. Related Work

We group related work into three categories: resource management, energy accounting, and energy efficiency.

8.1 Resource Management

Cinder's taps and reserves build on the abstraction of resource containers [Banga 1999]. Like resource containers, they provide a platform for attributing resource consumption to a specific principal. By separating resource management into rates and quantities, however, Cinder allows applications to delegate with reserves, yet reclaim unused resources. This separation also makes policy decisions much easier. Since resource containers serve both as limits and reservations, hierarchical composition either requires a single policy (limit or reserve) or ad hoc rules (a guaranteed CPU slot cannot be the child of a CPU usage limit).

Linux has recently incorporated "cgroups" [Menage 2008] into the mainline kernel, which are similar to resource containers, but group processes rather than threads. They are hierarchical and rely on "subsystem" modules that schedule particular resources (CPU time, CPU cores, memory).

ECOSystem [Zeng 2002, 2003] presents an abstraction for energy, "currentcy", which unifies a system's device power states. It represents logical tasks using a flat form of resource containers [Banga 1999] by grouping related processes in the same container. This flat approach makes it impossible for an application to delegate, as it must either share its container with a child or put it in a new container that competes for resources. Like ECOSystem, Cinder estimates energy consumption with a software-based model that ties runtime power states to power draw.

ECOSystem achieves pooling similar to Cinder's `netd` for devices with non-linear power consumption (disk and network access), using unique cost models for each device. Cinder simplifies construction of these policies using its fine-grained protection mechanism and reserves to provide the same result in userspace.

8.2 Measurement, Modeling, and Accounting

Accurately estimating a device's energy consumption is an ongoing area of research. Early systems, such as ECOSystem [Zeng 2002], use a simple linear combination of device states. Most modern phone operating systems, such as Symbian and OS X, follow this approach.

PowerScope improves CPU energy accuracy by correlating instrumented traces of basic blocks with program execution [Flinn 1999b]. A more recent system, Koala, explores how modern architectures can have counter-intuitive energy/performance tradeoffs, presenting a model based on performance counters and other state [Snowdon 2009]. A Koala-enabled system can use these estimates to specify a range of policies, including minimizing energy, maximizing performance, and minimizing the energy-delay product. The Mantis system achieves similar measurement accuracy to Koala using CPU performance counters [Economou 2006].

Quanto [Fonseca 2008] extends the TinyOS operating system to support fine-grained energy accounting across activities. Using a custom measurement circuit, Quanto generates an energy model of a device and its peripherals using a linear regression of power measurements. By monitoring the power state of each peripheral and dynamically tracking which activity is active, Quanto can give precise breakdowns of where a device is spending energy.

PowerBooter and PowerTutor [Zhang 2010] explore the generation of detailed power models for a full-featured smartphone (the HTC Magic) providing application power consumption estimation and feedback for tuning.

Cinder complements this work on modeling and accounting. Improved hardware support to determine where energy is going would make its accounting and resource control more accurate. On top of these models, Cinder provides a pair of abstractions that allow applications to flexibly and easily enforce a range of policies.

8.3 Energy Efficiency

There is rich prior work on improving the energy efficiency of individual components, such as CPU voltage and frequency scaling [Flautner 2002; Govil 1995], spinning down disks [Douglis 1995; Helmbold 1996], or carefully selecting memory pages [Lebeck 2000]. Phone operating systems today tend to depend on much simpler, but still effective optimization schemes than in the research literature, such as hard timeouts for turning off devices. The exact models or mechanisms used for energy efficiency are orthogonal to Cinder: they allow applications to complete more work within a given power budget. The image viewer described in §5.3 is an example of an energy-adaptive application, as is typical in the Odyssey system [Flinn 1999a].

9. Future Work

We believe that the reserve and tap abstractions may be fruitfully applied to other resource allocation problems beyond energy consumption. For instance, the high cost of mobile data plans makes network bits a precious resource. Applications should not be able to run up a user's bill due to expensive data tariffs, just as they should not be able to run down the battery unexpectedly. Since data plans are frequently offered in terms of megabyte quotas, Cinder's mechanisms could be repurposed to limit application network access by replacing the logical battery with a pool of network bytes. Similarly, reserves could also be used to enforce SMS text message quotas.

Using the HTC Dream's limited battery level information Cinder could adapt its energy model based on past component and application usage, dynamically refining its costs. Though Cinder can facilitate this, and we have made some adjustments to test this, evaluating the complex and dynamic system this would yield will require additional research.

10. Conclusion

Cinder is an operating system for modern mobile devices. It uses techniques similar to existing systems to model device energy use, while going beyond the capabilities of current operating systems by providing an IPC system that fundamentally accounts for resource usage on behalf of principals. It extends this accounting to add subdivision and delegation, using its reserve and tap abstractions. We have described and applied this system to a variety of applications demonstrating, in particular, their ability to partition applications to energy bounds even with complex policies. Additionally, we showed Cinder facilitates policies which enable efficient use of expensive peripherals despite non-linear power models.

Acknowledgments

We thank John Ousterhout, the anonymous reviewers, and our shepherd, Liuba Shrira, for their feedback. This work was supported by generous gifts from DoCoMo Capital, the National Science Foundation under grants #0831163, #0846014, and #0832820 POMI (Programmable Open Mobile Internet) 2020 Expedition Grant, the King Abdullah University of Science and Technology (KAUST), Microsoft Research, T-Mobile, NSF Cybertrust award CNS-0716806, and an NSERC Post Graduate Scholarship. This research was performed under an appointment to the U.S. Department of Homeland Security (DHS) Scholarship and Fellowship Program, administered by the Oak Ridge Institute for Science and Education (ORISE) through an interagency agreement between the U.S. Department of Energy (DOE) and DHS. ORISE is managed by Oak Ridge Associated Universities (ORAU) under DOE contract number DE-AC05-06OR23100. All opinions expressed in this paper are the authors' and do not necessarily reflect the policies and views of DHS, DOE, or ORAU/ORISE.

References

[Com 1988] THE EXECUTIVE COMPUTER; Compaq Finally Makes a Laptop. http://www.nytimes.com/1988/10/23/business/the-executive-computer-compaq-finally-makes-a-laptop.html, 1988.

[Fla 2009] Adobe and HTC Bring Flash Platform to Android, June 2009. http://www.adobe.com/aboutadobe/pressroom/pressreleases/pdfs/200906/062409AdobeandHTC.pdf.

[App 2010] Apple Previews iPhone OS 4, April 2010. http://www.apple.com/pr/library/2010/04/08iphoneos.html.

[Banga 1999] Gaurav Banga, Peter Druschel, and Jeffrey C. Mogul. Resource containers: a new facility for resource management in server systems. In *Proceedings of the 3rd Symposium on Operating Systems Design and Implementation*, pages 45–58, New Orleans, LA, 1999.

[Douglis 1995] Fred Douglis, P. Krishnan, and Brian N. Bershad. Adaptive Disk Spin-down Policies for Mobile Computers. In *Proceedings of the 2nd Symposium on Mobile and Location-Independent Computing*, pages 121–137, 1995.

[Economou 2006] Dimitris Economou, Suzanne Rivoire, and Christos Kozyrakis. Full-system power analysis and modeling for server environments. In *Proceedings of the 2nd Workshop on Modeling, Benchmarking and Simulation*, Boston, MA, 2006.

[Flautner 2002] Krisztian Flautner and Trevor Mudge. Vertigo: automatic performance-setting for linux. In *Proceedings of the 5th Symposium on Operating Systems Design and Implementation*, pages 105–116, Boston, MA, 2002.

[Flinn 1999a] Jason Flinn and M. Satyanarayanan. Energy-aware adaptation for mobile applications. In *Proceedings of the 17th ACM Symposium on Operating Systems Principles*, pages 48–63, Charleston, SC, 1999.

[Flinn 1999b] Jason Flinn and M. Satyanarayanan. PowerScope: A Tool for Profiling the Energy Usage of Mobile Applications. In *Proceedings of the 2nd IEEE Workshop on Mobile Computer Systems and Applications*, New Orleans, LA, 1999.

[Fonseca 2008] Rodrigo Fonseca, Prabal Dutta, Philip Levis, and Ion Stoica. Quanto: Tracking Energy in Networked Embedded Systems. In *Proceedings of the 8th Symposium on Operating Systems Design and Implementation*, pages 323–338, 2008.

[Govil 1995] Kinshuk Govil, Edwin Chan, and Hal Wasserman. Comparing algorithm for dynamic speed-setting of a low-power CPU. In *Proceedings of the 1st Conference on Mobile Computing and Networking*, pages 13–25, Berkeley, CA, 1995.

[Helmbold 1996] David P. Helmbold, Darrell D. E. Long, and Bruce Sherrod. A dynamic disk spin-down technique for mobile computing. In *Proceedings of the 2nd Conference on Mobile Computing and Networking*, pages 130–142, Rye, NY, 1996.

[Lebeck 2000] Alvin R. Lebeck, Xiaobo Fan, Heng Zeng, and Carla Ellis. Power aware page allocation. In *Proceedings of the 9th International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 105–116, Cambridge, MA, 2000.

[Menage 2008] Paul Menage. cgroups, October 2008. http://git.kernel.org/?p=linux/kernel/git/torvalds/linux-2.6.git;a=blob;f=Documentation/cgroups/cgroups.txt;hb=b851ee7921fabdd7dfc96ffc4e9609f5062bd12.

[Snowdon 2009] David C. Snowdon, Etienne Le Sueur, Stefan M. Petters, and Gernot Heiser. Koala: a platform for OS-level power management. In *Proceedings of the 4th ACM European Conference on Computer Systems*, pages 289–302, Nuremberg, Germany, 2009.

[Waldspurger 2002] Carl A. Waldspurger. Memory resource management in VMware ESX server. *SIGOPS Oper. Syst. Rev.*, 36: 181–194, December 2002. ISSN 0163-5980.

[Zeldovich 2006] Nickolai Zeldovich, Silas Boyd-Wickizer, Eddie Kohler, and David Mazières. Making information flow explicit in HiStar. In *Proceedings of the 7th Symposium on Operating Systems Design and Implementation*, pages 263–278, Seattle, WA, 2006.

[Zeng 2002] Heng Zeng, Carla S. Ellis, Alvin R. Lebeck, and Amin Vahdat. ECOSystem: managing energy as a first class operating system resource. In *Proceedings of the 10th International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 123–132, San Jose, CA, 2002.

[Zeng 2003] Heng Zeng, Carla S. Ellis, Alvin R. Lebeck, and Amin Vahdat. Currentcy: A unifying abstraction for expressing energy management policies. In *Proceedings of the 2003 USENIX Annual Technical Conference*, pages 43–56, San Antonio, TX, 2003.

[Zhang 2010] Lide Zhang, Birjodh Tiwana, Zhiyun Qian, Zhaoguang Wang, Robert P. Dick, Zhuoqing Morley Mao, and Lei Yang. Accurate online power estimation and automatic battery behavior based power model generation for smartphones. In *Proceedings of the eighth IEEE/ACM/IFIP international conference on Hardware/software codesign and system synthesis*, CODES/ISSS '10, pages 105–114, New York, NY, USA, 2010. ACM. ISBN 978-1-60558-905-3.